Rapid Performance Re-engineering of Distributed Embedded Systems via Latency Analysis and K-Level Diagonal Search ⋆

Jungkeun Park [a], Minsoo Ryu [b,*], Seongsoo Hong [a], Lucia Lo Bello [c]

[a] *School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-742, Korea.*

[b] *College of Information and Communications, Hanyang University, Seoul 133-791, Korea.*

[c] *Department of Computer Engineering and Telecommunications, University of Catania, Catania, Italy.*

Abstract

This paper presents a systematic methodology aimed at rapid and cost-effective re-engineering of distributed embedded systems. We define embedded system re-engineering as an analysis and alteration of a legacy system to guarantee newly imposed performance requirements such as throughput and input-to-output latency. Our methodology pinpoints performance bottlenecks of a system and selectively upgrades processing elements at the least cost. Inputs for our methodology include a system design specified by a process network over a set of processing elements and a new throughput requirement. The output is a set of scaling factors that represent the ratios of the performance upgrades for processing elements.

Preprint submitted to Elsevier Science *5 January 2005*

Our methodology works in two steps. First, it estimates the latency of each process and identifies bottleneck processes. Second, it derives a system of constraints with scaling factors being free variables and formulates an optimization problem. Then, it solves the optimization problem for scaling factors with an objective of minimizing upgrade cost. For this methodology, we propose an accurate latency analysis technique for precedence-constrained tasks under preemptive fixed priority scheduling. We also propose a k-level diagonal search algorithm that allows us to trade optimality for search time. Our experimental results show the effectiveness of the proposed re-engineering approach.

Key words: distributed embedded systems, re-engineering, latency analysis, performance/cost optimization

---

⋆ The work reported in this paper was supported in part by MOST under the National Research Laboratory (NRL) grant M1-9911-00-0120, by the Institute of Computer Technology (ICT), by the Automation and Systems Research Institute (ASRI), and by the research fund of Hanyang University (HY-2003-T). An earlier version of this paper appeared in preliminary form in the *Proceedings of IEEE Real-Time Systems Symposium*, (December 2001).

* Corresponding author.

Email addresses: jkpark@redwood.snu.ac.kr (Jungkeun Park), msryu@hanyang.ac.kr (Minsoo Ryu), sshong@redwood.snu.ac.kr (Seongsoo Hong), llobello@diit.unict.it (Lucia Lo Bello).

1 Introduction

Due to the diversity and complexity of embedded systems and due to increased competition in the associated industry, developers are under very stringent requirements for increasing production speed. Many techniques and methodologies have been proposed to assist them in designing, analyzing, and testing embedded systems [19,9,21,16,26,22]. Recently, component-based software design approaches have been widely adopted for the rapid development of application-specific embedded systems.

In these approaches, an embedded system can be prototyped by composing reusable components. Such a development prototype is often subject to design modification when it fails to meet a given performance specification. In that case, the developer should locate performance bottlenecks in the prototype system, explore design alternatives using component libraries, and replace the bottleneck components with new ones at the least cost. Similar problems are encountered in industry during the re-engineering of a legacy system, when a product with additional features and enhanced performance is developed by modifying an old design.

Generally, the re-engineering problem is defined as a sequence of activities involving reverse engineering, system alteration, and forward engineering [9]. During a re-engineering process, the reverse engineering captures an understanding of the behavior and structure of the system, the system alteration modifies the structure and components of the system for enhanced performance, and the forward engineering creates new functionalities. Using this terminology, we define the performance re-engineering of an embedded sys-

tem as a specific instance of the re-engineering problem, such that the reverse engineering corresponds to the bottleneck process analysis within the system, and the system alteration performs the latency reduction of the system. Such a performance re-engineering problem is of the utmost practical importance during the production of embedded systems, since it can lead to significant reduction in development time and cost.

Unfortunately, the performance re-engineering problem for an embedded system poses serious challenges to developers. First, it is quite difficult to accurately estimate the latency of an embedded system, since this requires extensive static timing analysis of the system. Because embedded systems often consist of a network of processes that run on a heterogeneous distributed multiprocessor platform composed of microprocessors, microcontrollers, digital signal processors, and application-specific instruction set processors, the complexity of this task is considerable. Second, it is fairly difficult to eliminate performance bottlenecks in the system since system resources are shared in a complicated manner, thus minor changes in a single processor may affect the synchronization and timing behavior of the entire system.

While there exist plenty of design techniques and software tools for embedded systems which are based on real-time scheduling theory and formal methods [8,5,20,18,2,6,7,25,22], relatively few approaches address the performance re-engineering aspect of embedded systems. Without the help of systematic re-engineering methodologies, developers often resort to the ad hoc iteration of system analysis and re-design that often leads to over-optimization of the system. It is fairly obvious that this approach will fail when the system to be re-engineered becomes complex.

In this paper, we present a systematic methodology that allows rapid and cost-effective re-engineering of distributed embedded systems. A distributed embedded system is modeled as a process network and task graphs, where tasks are executed by a priority-based preemptive scheduler, as in many embedded systems. A performance requirement is given as the throughput of the system. For rapid performance re-engineering, our approach attempts to upgrade only the processing elements that execute bottleneck processes, while leaving the architecture and implementation intact. Inputs to our re-engineering problem are as follows:

(1) A process network and task graphs representing the underlying system.
(2) Task allocation and priority assignment.
(3) A desired throughput requirement.
(4) Hardware upgrade cost tables at various performance profiles.

With the above inputs, the objective of our approach is to find speedup ratios of processing elements that satisfy the new throughput requirement with minimal hardware upgrade costs. Our approach is based on latency analysis and a cost-benefit optimization. We identify performance bottlenecks of the system by estimating the latency of each process and eliminate such bottlenecks by formulating and solving an optimization problem. To do so, our approach employs the following two techniques:

(1) An accurate latency analysis technique that estimates the latency of each process.
(2) An effective heuristic search algorithm that solves the optimization problem.

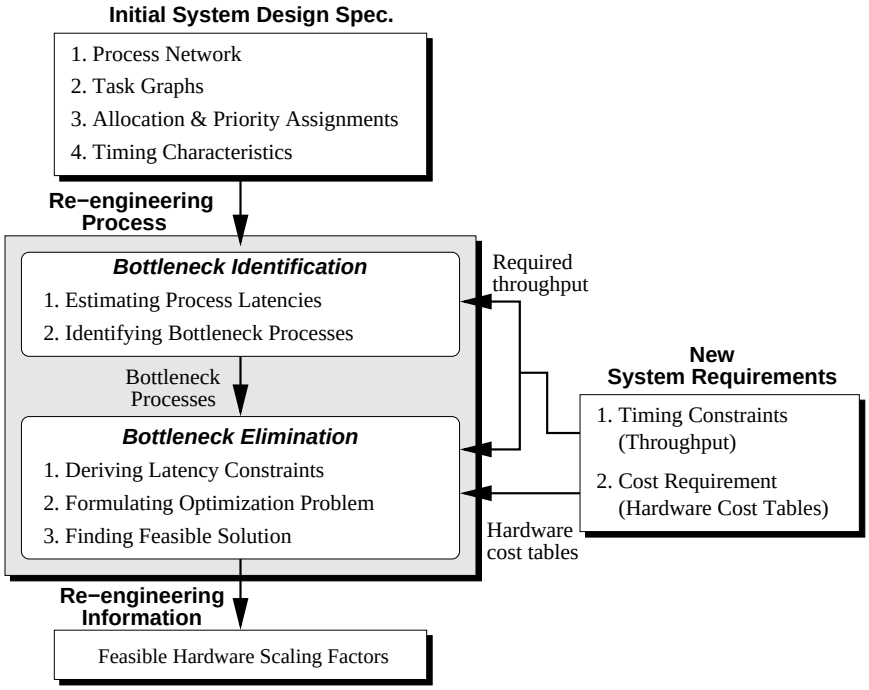


Fig. 1. Overview of the approach.

The proposed re-engineering method works in two steps, *bottleneck identification*, and *bottleneck elimination*, as shown in Figure 1. First, it estimates the latency of each process and identifies bottleneck processes. Second, it derives a set of latency constraints for each bottleneck process and formulates an optimization problem with an objective of minimizing the re-engineering cost. Then, it finds optimal speedup ratios for processing elements that need speedups to improve the performance of bottleneck processes.

1.1 Related Work

Existing re-engineering methods primarily deal with functional and structural analysis and modification of software systems [19,11,9] and hardware systems [16,26]. Madisetti et al. [16] propose a systematic technique for rapidly upgrading electronic systems. They propose using virtual prototyping accompanied

by their tools and libraries of simulatable models. Their approach is to evaluate the cost and benefit of re-engineering while performing hardware/software cosimulation. To facilitate electronic hardware re-engineering, Tummala and Madisetti [26] show that the SoP (System on a Package) paradigm provides more architectural flexibility than the SoC (System on a Chip) paradigm, thus enabling rapid re-engineering via reusable libraries. The re-engineering approaches in [16,26] are similar to ours in the sense that they adopt hardware upgrades as a way of re-engineering.

We are focusing on performance re-engineering of distributed embedded systems which requires strict latency analysis and efficient latency reduction techniques. Since estimating accurate process latencies for distributed systems is generally NP-hard as proven in [27], several heuristic algorithms were proposed in the literature [23,4,24,27]. Tindell el al. [24] analyzed worst-case latencies in tasks that are under preemptive fixed priority scheduling in distributed embedded systems by extending time demand analysis for single processor systems [13]. Sun [23] modeled precedence-constrained tasks as a chain of tasks and proposed a schedulability analysis based on time demand analysis in order to bound the latency of each task. However, his application model does not allow for precedence constraints between two tasks in different task chains. Our application model allows specifying general precedence constraints among tasks in the form of a task graph where arbitrary but acyclic precedence relationships can exist between any tasks. Yen and Wolf [27] considered the same application model as ours and proposed an iterative method based on *separation analysis* to compute upper bounds on the input-to-output latencies of processes. While our analysis is based on Yen and Wolf's separation analysis,

it yields tighter bounds by introducing a new technique called *interference time analysis*.

For latency reduction, most research attempts to reduce the schedule length of critical paths by changing the task allocations and the schedule. Ahmad and Kwok [1] propose an algorithm that can reduce the input-to-output latency of a task graph that runs on a parallel and distributed platform via static scheduling. The key idea of the algorithm is to reduce the schedule length of the critical path in the system by duplicating selected tasks and allocating them to other processing elements. This algorithm has an important weakness in that it attempts to manipulate only the critical path. This may result in excessive schedule length reduction and make other paths become new critical paths. Unlike this approach, ours performs global optimization of all the paths in the system and does not result in costly over-optimization.

Existing hardware/software co-synthesis approaches also address performance/cost optimization problems similar to ours [28,29]. One of the major issues in co-synthesis is the decision to either map functionalities into dedicated hardware or to implement them on a general-purpose microprocessor. This choice, known as hardware/software partitioning problem, should be based on achievable performance estimation and implementation cost. However, our re-engineering approach does not consider the movement of functionalities between hardware and software, but focuses on straightforward hardware upgrades while maintaining the original mapping of functionalities. The rationale behind this is that many industrial product series are released with identical hardware architectures and a high level of code reuse.

Although we have not considered the hardware/software partitioning issue in this work, we believe that our approach would be very useful during that stage of hardware/softwar co-design. Once an initial design has been made, our latency analysis can be used for precise performance estimation and our algorithm can be used to determine the required level of performance increase.

This paper provides major extensions to our earlier work [17] in both theoretical and practical directions. Most importantly, we eliminate the inflexible scheduling assumptions made in [17]. The previous approach [17] assumed static and non-preemptive scheduling to make timing analysis simple. This seriously restricted the applicability of the previous approach since many existing real-time systems use priority-based preemptive scheduling policies to achieve increased flexibility. In this paper, we adopt fixed priority preemptive scheduling and present a new timing analysis method to estimate process latencies. In addition, we mathematically analyze the proposed algorithm. We present a theorem that describes an essential property of the algorithm and then proceed to prove the theorem.

This paper is organized as follows. Section 2 defines the application model of a distributed embedded system along with its timing and performance constraints. Section 3 presents the latency analysis and the identification of bottleneck processes. Section 4 describes the derivation of linear latency constraints for bottleneck processes and the formulation of an optimization problem. Section 5 describes our heuristic algorithm for solving the optimization problem. Section 6 illustrates some experimental results on the proposed algorithm to show its effectiveness. Section 7 states our conclusions.

2 System Model

In this section, we present an application model consisting of a process network and task graphs, along with timing characteristics associated with the application model. As a walk-through example throughout the paper, we have chosen a digital copier since it possesses the timing constraints and design problems of a typical distributed embedded system. We specify it with the presented application model.

2.1 Application Model

As in many other embedded system models, we use a graphical model with hierarchical abstraction [12]. Our framework renders a distributed embedded system as both a process network (PN) and a set of task graphs. Our process network is similar to Kahn's process network [10], which is a computational model where a number of concurrent processes communicate through unidirectional FIFO channels [12]. In this paper, we generalize the notion of Kahn's process to support electro-mechanical functions as well as computational functions. Our process network model is a direct acyclic graph $G(\mathcal{P}, E)$ such that

- $\mathcal{P} = \{\sigma_1, \ldots, \sigma_u\}$ is a set of processes. A process is a transformation of one or more inputs to outputs of another form, where the inputs and outputs represent computational data or physical objects.
- $E \subseteq \mathcal{P} \times \mathcal{P}$ is a set of directed edges such that $\sigma_i \rightarrow \sigma_j$ denotes precedence from σ_i to σ_j. Each process starts after it accepts inputs from all of its immediate predecessors.

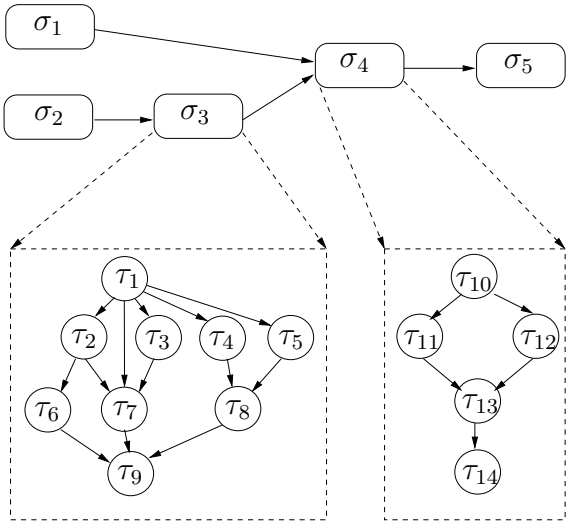


Fig. 2. Process network and task graph.

Once initiated, a process consumes time, thus causing input-to-output latency. The latency may include the additional time caused by the interference from other processes due to the sharing of common hardware resources such as processors. Also, each process has a period that is common among all the processes in the system. Our process network model is similar to processor pipelining where a process can be mapped to a single pipeline stage. Each process runs in parallel with other processes producing outputs at a common rate.

As shown in Figure 2, a process in a process network can be expanded into a task graph, like processes σ_3 and σ_4. A task graph is given as a direct acyclic graph $G(V, E')$ such that

- $V = \{\tau_1, \ldots, \tau_v\}$ is a set of tasks in a process.
- $E' \subseteq V \times V$ is a set of directed edges such that $\tau_i \rightarrow \tau_j$ denotes precedence from τ_i to τ_j. Edges and tasks in a task graph have exactly the same semantics as those in a process network.

We consider a generic hardware platform that consists of heterogeneous processing elements $PE = \{\pi_1, \ldots, \pi_n\}$, which include microprocessors, microcontrollers, DSPs, FPGA, ASIC, and electro-mechanical components. With this hardware model, we assume that tasks have been partitioned and statically allocated to processing elements. Such task allocation is denoted by Π that is a mapping of tasks onto the set PE such that $\Pi : V \mapsto PE$.

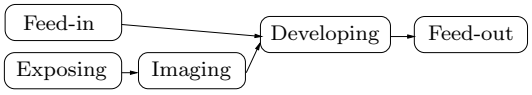
As mentioned before, our major objective is to leave the system architecture and implementation as intact as possible. The rationale behind this is that many industrial product series are released with identical hardware architectures and a high level of code reuse. Hence, our re-engineering approach relies on straightforward upgrades of processing elements without remapping tasks onto heterogeneous processing elements. For example, if some tasks are found to be performance bottlenecks on a 200 MHz microprocessor, we merely replace the processor with a 300 or 400 MHz version to gain a required performance increase. To represent such various performance increase options, each processing element π_i is associated with the scaling factor $\mathcal{S}_i$ that denotes the scaled performance of π_i. If there are m_i options in upgrading π_i, the scaling factor $\mathcal{S}_i$ can take m_i discrete values $\mathcal{S}_{i,1}, \mathcal{S}_{i,2}, \ldots, \mathcal{S}_{i,j}, \ldots, \mathcal{S}_{i,m_i}$ where $\mathcal{S}_{i,1} < \mathcal{S}_{i,2} < \ldots < \mathcal{S}_{i,j} \ldots < \mathcal{S}_{i,m_i}$ and $\mathcal{S}_{i,m_i} = 1.0$. The unit scaling factor $\mathcal{S}_i = 1.0$ denotes the current processing element and $\mathcal{S}_i < 1.0$ denotes a faster one. For each $\mathcal{S}_i$ of π_i, the cost function $c_i(\mathcal{S}_i)$ is associated and $c_i(\mathcal{S}_i)$ is a decreasing function of $\mathcal{S}_i$. Thus, when a certain scaling factor $\mathcal{S}_{i,j_i}$ is chosen for processing element π_i, the total hardware cost of the system is the sum of $c_i(\mathcal{S}_{i,j_i})$: $\sum_{\pi_i \in PE} c_i(\mathcal{S}_{i,j_i})$.

Note that we may introduce new hardware implementations using FPGA or ASIC into the legacy hardware platform. For instance, we can partition the tasks on a bottleneck processor into a software part that will be kept running on the processor and a hardware part that will be implemented through FPGA or ASIC. However, we do not consider this approach in this work because it requires a significant architectural change and also prevents the reuse of legacy software code.

Each task can be released only after all of its predecessors complete. If co-allocated tasks are simultaneously ready on the same processing element, the processing element always executes the highest-priority ready task on a fixed-priority basis. By $\tau_i \succ \tau_j$, we mean that τ_i's priority is higher than τ_j's priority. Finally, the execution time e_i of task τ_i is modeled by a bounded interval $[e_i^{lower}, e_i^{upper}]$ due to factors such as conditional behavior and the inaccuracy of the WCET analysis techniques [3,14].

2.2 A Digital Copier Example

To show the expressive power of our application model, we specify a digital copier in the PN format. A digital copier is a typical example of a distributed embedded system that possesses various electro-mechanical components and strict timing constraints. Its major components include a scanner, a laser-beam printer, an organic photo-conductive (OPC) drum, paper feeders, and transfer belts. The entire copying process can be broken down into five subprocesses – feed-in, exposing, imaging, developing, and feed-out – as described in [17]. Figure 3 shows the PN of all copy processes and their timing characteristics.



(A)

process	latency	period
Feed-in (σ_1)	0.3	2.0
Exposing (σ_2)	0.5	2.0
Imaging (σ_3)	2.0	2.0
Developing (σ_4)	1.6	2.0
Feed-out (σ_5)	0.3	2.0

(B)

Fig. 3. (A) Process network and (B) timing characteristics of digital copier.

The throughput of the digital copier is defined as the number of copies per minute and is determined by the common period of processes. From Figure 3(B), we see that the imaging process takes the longest time, 2s: it involves compute-intensive digital image processing. In this example, the period is thus 2s and throughput is $60/2 = 30$ cpm. If we want to improve the throughput up to 40 cpm, we must reduce the latency of each process down to $60/40 = 1.5$s.

3 Latency Analysis and Bottleneck Process Identification

In this section, we first describe a timing analysis for a given process network model. Our timing analysis focuses on estimating the input-to-output latency of a process that consists of periodic, precedence-constrained tasks under fixed-priority assignment. We then show how to determine bottleneck processes by using the result of the timing analysis.

3.1 Estimating Process Latencies

The latency of a process is defined as the time from the instant it is invoked to the instant it completes. Since each process consists of precedence-constrained tasks, its latency can also be viewed as the duration of time from the invocation

of its first task to the completion of its last task. Thus, our analysis attempts to compute the start and finish time of each task under precedence constraints, and then determines the input-to-output latency of the process. Let s_i and f_i be the start and finish time of task τ_i, respectively. To keep notations simple, we define s_i and f_i to be relative values from the start of the period of the process σ_j that possesses τ_i. According to this definition, if τ_i is the first task of process σ_j, the start time of τ_i is $s_i = 0$. Let $Pred(\tau_i)$ be the set of tasks that are predecessors of τ_i in a given task graph. The analysis begins with the following recursive equation for task τ_i, which captures precedence relationships among tasks and time demand generated by equal or higher priority tasks including itself.

$$f_i = s_i + I_i + e_i \tag{1}$$

where $s_i = \max_{\tau_j \in Pred(\tau_i)}\{f_j\}$ and I_i is the interference time generated by equal or higher priority tasks on the same processing element $\Pi(\tau_i)$. Note that if we know the interference time I_i for each τ_i in the above, we can determine the input-to-output latency of a process by recursively applying the above equation Eq.(1). Thus in the following discussion, we focus on finding a tight upper bound on the interference time I_i.

To derive a tight bound on I_i, we iteratively apply two techniques, *separation analysis* and *interference time analysis*. The separation analysis determines which tasks can interfere with the target task τ_i, for which we are trying to bound the interference time I_i. The interference time analysis then attempts to accurately compute the delay caused by each interfering task.

Separation Analysis: Let I_i^{upper} be the upper bound on the interference time I_i for τ_i. A simple and safe computation of I_i^{upper} is to take the sum of the execution times of all the independent equal or higher priority tasks on the same processing element $\Pi(\tau_i)$.

$$I_i^{upper} = \sum_{\tau_j \in \Phi(\tau_i)} e_j^{upper} \tag{2}$$

$$\Phi(\tau_i) = \{\tau_j | \tau_j \succeq \tau_i, \Pi(\tau_j) = \Pi(\tau_i), \tau_i \not\leftrightarrow \tau_j\} \tag{3}$$

where $\tau_i \not\leftrightarrow \tau_j$ denotes $\tau_j \notin Pred(\tau_i)$ and $\tau_i \notin Pred(\tau_j)$. However, we observe that some equal or higher priority tasks τ_j cannot interfere with τ_i if they can never be activated simultaneously. For instance, it is not possible for τ_j to delay τ_i if τ_j always completes before τ_i starts or τ_j always starts after τ_i completes.

To determine which tasks can delay τ_i, we introduce the notion of *worst-case execution window* for each task. Let s_i^{lower} and f_i^{upper} be the lower bound on start time and the upper bound on finish time of task τ_i, respectively. The worst case execution window W_i of τ_i is then defined as the time interval $[s_i^{lower}, f_i^{upper}]$, within which τ_i starts and completes.

$$s_i^{lower} = \max_{\tau_j \in Pred(\tau_i)} \{f_j^{lower}\} \tag{4}$$

$$f_i^{upper} = s_i^{upper} + I_i^{upper} + e_i^{upper} \tag{5}$$

where $f_j^{lower} = s_i^{lower} + e_i^{lower}$ and $s_i^{upper} = \max_{\tau_j \in Pred(\tau_i)}\{f_j^{upper}\}$. Here, the lower bound s_i^{lower} can be interpreted as the earliest possible start time of τ_i under precedence constraints. The upper bound f_i^{upper} can be interpreted as the latest possible finish time of τ_i.

Initially, the worst-case execution window W_i can be obtained by applying equation $\Phi(\tau_i) = \{\tau_j | \tau_j \succeq \tau_i, \Pi(\tau_j) = \Pi(\tau_i), \tau_i \not\leftrightarrow \tau_j\}$ to set $\Pi(\tau_j)$. We then repeatedly check for each task τ_i whether W_j and W_i overlap or not. If W_j and W_i do not overlap, we can eliminate τ_j from the interfering task set $\Phi(\tau_i)$. To eliminate non-overlapping tasks, we define the set of interfering tasks as follows:

$$\Phi(\tau_i) = \{\tau_j | f_j^{upper} \geq s_i^{lower}, f_i^{upper} \geq s_j^{lower}, \tau_j \succeq \tau_i, \Pi(\tau_j) = \Pi(\tau_i), \tau_i \not\leftrightarrow \tau_j\} \tag{6}$$

where $f_j^{upper} \geq s_i^{lower}$ and $f_i^{upper} \geq s_j^{lower}$ mean that W_j and W_i do not overlap.

Interference Time Analysis: This analysis attempts to give an accurate bound on the interference time I_i caused by $\Phi(\tau_i)$, which is determined by the separation analysis. An important drawback of the computation of I_i^{upper} in Eq.(2) is that it conservatively takes the sum of the bounds e_j^{upper}, thus resulting in unsatisfactory bounds on process latencies. However, in many cases, we observe that the actual delay contributed by $\tau_j \in \Phi(\tau_i)$ can be less than the maximum execution time e_j^{upper} depending on the phasing of execution windows and the length of window overlap. Based on this observation, the interference time analysis derives tighter bounds on the interference times than the separation analysis alone.

Figure 4 illustrates the need for the interference analysis in two cases of window phasing. Example (A) of Figure 4 shows the first case where the execution window of higher priority task τ_j starts earlier than that of lower priority task τ_i. In this example, we see that the length of window overlap is 3 and the

maximum execution time e_j^{upper} of τ_j is 7. Since τ_j completes by 13 and τ_i can start only after 10, τ_j can delay τ_i at most by 3, not by the $e_j^{upper} = 7$. The example (B) of Figure 4 shows the opposite case where the execution window of higher priority task τ_j starts later than that of lower priority task τ_i. Our observation is that if e_j^{upper} is more than the length of window overlap, then τ_j never preempts τ_i. We can prove this by contradiction. Suppose that τ_j preempts τ_i at time 10 at which τ_i was executing. Since τ_j can delay τ_i by $e_j^{upper} = 4$, τ_i may complete later than $f_i^{upper} = 13$. This is a contradiction since $f_i^{upper} = 13$ is the upper bound on the finish time of τ_i.

Let $I_{i,j}^{upper}$ for τ_i be the upper bound on the delay contributed by $\tau_j \in \Phi(\tau_i)$. Our interference time analysis can be described by

$$I_i^{upper} = \sum_{\tau_j \in \Phi(\tau_i)} I_{i,j}^{upper} \tag{7}$$

where

$$\text{Case A } (s_j^{lower} < s_i^{lower})\text{: } I_{i,j}^{upper} = \begin{cases} e_j^{upper} & \text{if } e_j^{upper} < f_j^{upper} - s_i^{lower} \\ f_j^{upper} - s_i^{lower} & \text{otherwise} \end{cases} \tag{8}$$

$$\text{Case B } (s_j^{lower} \geq s_i^{lower})\text{: } I_{i,j}^{upper} = \begin{cases} e_j^{upper} & \text{if } e_j^{upper} < f_i^{upper} - s_j^{lower} \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

Walk-through Example: Consider the digital copier example described in Section 2.2. As an illustration, we derive the upper bound on finish time of τ_{11}

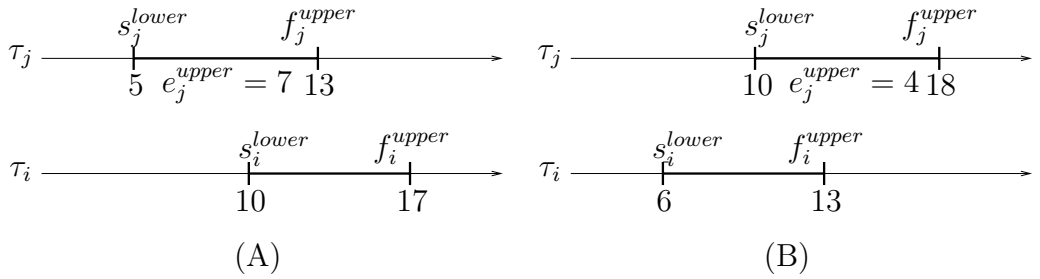


Fig. 4. Interference time analysis examples: (A) W_j starts earlier than W_i ($s_j^{lower} < s_i^{lower}$) and (B) W_j starts equal to or later than W_i ($s_j^{lower} \geq s_i^{lower}$).

by applying the proposed analysis techniques. Task allocation with priority assignment is given in Table 1 and execution times are given in Table 2. Relevant task graphs are given in Figure 5 where the shaded area includes all the tasks that share the same processing element π_1 with τ_{11}.

Initially, the interfering task set Φ_{11} for τ_{11} includes all the higher priority tasks so that $\Phi_{11} = \{\tau_1, \tau_3, \tau_5\}$. Hence, the initial value of I_{11}^{upper} is $e_1^{upper} + e_3^{upper} + e_5^{upper} = 6$, thus giving the initial upper bound $f_{11}^{upper} = s_{11}^{upper} + I_{11}^{upper} + e_{11}^{upper}$. Since τ_{10} is the only predecessor of τ_{11}, we have $s_{11}^{lower} = f_{10}^{lower} = e_{10}^{lower} = 5$. Thus, it immediately follows that $W_{11} = [5, 14]$.

Similarly, we can obtain $W_1 = [0, 2]$, $W_3 = [1.5, 6]$, and $W_5 = [1.5, 4]$ for each task in $\Phi_{11} = \{\tau_1, \tau_3, \tau_5\}$. If we apply the separation analysis, we can eliminate τ_1 and τ_5 from the interfering task set Φ_{11} since their worst-case execution windows do not overlap with τ_{11}'s window. Thus, the interfering task set Φ_{11} becomes $\{\tau_3\}$ and I_{11}^{upper} is reduced to 2. This upper bound I_{11}^{upper} can be further tightened if we apply the interference time analysis. Since the length of window overlap between τ_3 and τ_{11} is $f_3^{upper} - s_{11}^{lower} = 6 - 5 = 1$ which is less than $e_3^{upper} = 2$, the interference time caused by τ_3 is reduced to 1. As a result, we have $f_{11}^{upper} = s_{11}^{upper} + I_{11}^{upper} + e_{11}^{upper} = 6 + 1 + 2 = 9$.

Table 3 shows the final results of the timing analysis for processes σ_3 and σ_4.

Table 1
Priority assignment for each processing elements.

processing element	allocated tasks and priority assignment
π_1	$\tau_1 \succ \tau_5 \succ \tau_3 \succ \tau_{11} \succ \tau_8 \succ \tau_{14} \succ \tau_9$
π_2	$\tau_4 \succ \tau_2 \succ \tau_6 \succ \tau_7$
π_3	$\tau_{10} \succ \tau_{12} \succ \tau_{13}$

Table 2
Task execution times.

task	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_9	τ_{10}	τ_{11}	τ_{12}	τ_{13}	τ_{14}
e_i^{lower}	1.5	2	1.5	2.5	1.5	2	1	2.5	0.5	5	1.5	1.5	1.5	0.5
e_i^{upper}	2	3	2	3	2	3	2	3	1	6	2	2	3	1

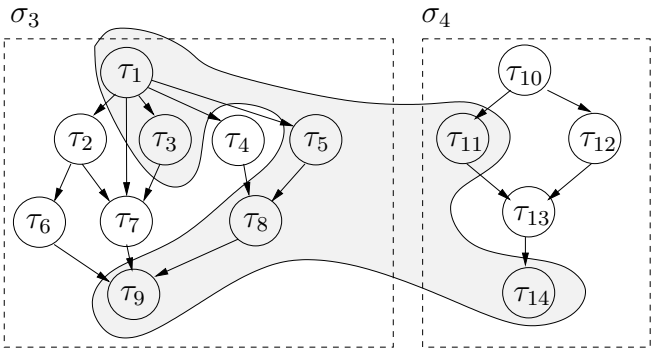


Fig. 5. Task graph of process σ_3 and σ_4.

Note that τ_9 and τ_{14} are the last tasks of σ_3 and σ_4, respectively. Thus, the latency of σ_3 is bounded by $f_9^{upper} = 20$ and the latency of σ_4 is bounded by $f_{14}^{upper} = 16$. By applying our timing analysis to each processing element of our walk-through example, we can obtain the results reported in Figure 3 (B).

3.2 Identifying Bottleneck Processes

We present the identification of bottleneck processes using the latency analysis results. For each process σ_i, let $\mathcal{L}_i$ be the latency bound that is given by our latency analysis. Also, let $\mathcal{L}$ be the system's latency constraint that is determined by its new throughput constraint. Then process σ_i is defined to be

Table 3
Timing analysis result of processes σ_3 and σ_4.

task	s_i^{lower}	s_i^{upper}	f_i^{lower}	f_i^{upper}	$\Phi(\tau_i)$	I_i	W_i
τ_1	0	0	1.5	2			[0, 2]
τ_2	1.5	2	3.5	8	τ_4	3	[1.5, 8]
τ_3	1.5	2	3	6	τ_5	2	[1.5, 6]
τ_4	1.5	2	4	5			[1.5, 5]
τ_5	1.5	2	3	4			[1.5, 4]
τ_6	3.5	8	5.5	14	τ_4	3	[3.5, 14]
τ_7	3.5	8	4.5	16	τ_4, τ_6	6	[3.5, 16]
τ_8	4	5	6.5	12	τ_3, τ_{11}	4	[4, 12]
τ_9	6.5	16	7	20	τ_{11}, τ_{14}	3	[6.5, 20]
τ_{10}	0	0	5	6			[0, 6]
τ_{11}	5	6	6.5	9	τ_3	1	[5, 9]
τ_{12}	5	6	6.5	8			[5, 8]
τ_{13}	6.5	9	8	12			[6.5, 12]
τ_{14}	8	12	8.5	16	τ_8	3	[8, 16]

a bottleneck process if its latency $\mathcal{L}_i$ is greater than the period (or latency) constraint $\mathcal{L}$. If we denote the set of bottleneck processes by $\mathcal{P}^{\mathcal{L}}_{bottleneck}$, it is defined by

$$\mathcal{P}^{\mathcal{L}}_{bottleneck} = \{\sigma_i \in \mathcal{P} | \mathcal{L}_i > \mathcal{L}\}.$$

Walk-through Example: Recall the digital copier example. Suppose that the new latency requirement $\mathcal{L}$ is 15 time units. From Figure 3 (B), we see that the imaging process (σ_3) and developing process (σ_4) are bottlenecks for the new design since they have the maximum latency of 20 and 16 time units respectively, which exceeds the required latency.

4 Optimization Problem Formulation

In this section, we first describe the derivation of linear latency constraints for bottleneck processes. Then, we present the formulation of an optimization problem with the objective of minimizing upgrade cost. The formulated optimization problem makes an integer programming problem that is NP hard.

4.1 Deriving Latency Constraints

Once bottleneck processes σ_k are identified, we need to reduce their latency bounds $\mathcal{L}_k$ to the required latency $\mathcal{L}$ by appropriately speeding up underlying processing elements. Since our goal is to minimize the speedup cost, we introduce performance scaling factors into our latency analysis and attempt to find cost-effective values for them.

In order to derive latency constraints, we first calculate the latency of each process with scaled execution times. Suppose that scaling factor $\mathcal{S}_p$ for processing element $\Pi(\tau_i)$ can take m_p discrete values $\mathcal{S}_{p,1} < \mathcal{S}_{p,2} < \ldots < \mathcal{S}_{p,q} < \ldots < \mathcal{S}_{p,m_p}(= 1)$, which are given by the cost table. Then, the scaled execution time of τ_i is represented by $e_i \cdot \mathcal{S}_p$. We calculate the latency of each process by replacing e_i^{upper} with $e_i^{upper} \cdot \mathcal{S}_p$ in the latency estimation in Section 3. Thus, the upper bounds on the start and finish times of τ_i are given by

$$s_i^{upper} = \max_{\tau_j \in Pred(\tau_i)} \{f_j^{upper}\}. \tag{10}$$

$$f_i^{upper} = s_i^{upper} + I_i^{upper} + e_i^{upper} \cdot \mathcal{S}_p \tag{11}$$

To calculate I_i^{upper} by applying the separation analysis and the interference

time analysis, we need to determine the constant worst-case execution window for each task. For this reason, we use the minimum execution time $e_i^{lower} \cdot \mathcal{S}_{p,1}$ and the maximum execution time $e_i^{upper} \cdot \mathcal{S}_{p,m_p}$ when determing worst-case execution windows. Then, the worst-case execution window is described by $\overline{W_i} = [\overline{s_i^{lower}}, \overline{f_i^{upper}}]$;

$$\overline{s_i^{lower}} = \max_{\tau_j \in Pred(\tau_i)} \{\overline{f_j^{lower}}\} \tag{12}$$

$$\overline{f_i^{upper}} = \overline{s_i^{upper}} + \overline{I_i^{upper}} + e_i^{upper} \cdot \mathcal{S}_{p,m_p} = \overline{s_i^{upper}} + \overline{I_i^{upper}} + e_i^{upper} \tag{13}$$

where $\overline{f_j^{lower}} = \overline{s_i^{lower}} + e_i^{lower} \cdot \mathcal{S}_{p,1}$ and $\overline{s_i^{upper}} = \max_{\tau_j \in Pred(\tau_i)} \{\overline{f_j^{upper}}\}$. $\overline{I_i^{upper}}$ is also calculated by the separation analysis and the interference time analysis with $e_i^{upper} \cdot \mathcal{S}_{p,m_p}$.

With this worst-case execution window, the separation analysis is described by

$$\Phi(\tau_i) = \{\tau_j | \overline{f_j^{upper}} \geq \overline{s_i^{lower}}, \overline{f_i^{upper}} \geq \overline{s_j^{lower}}, \tau_j \succeq \tau_i, \Pi(\tau_j) = \Pi(\tau_i), \tau_i \not\leftrightarrow \tau_j\} \tag{14}$$

Also, the interference time analysis is described by

$$\text{Case A } (\overline{s_j^{lower}} < \overline{s_i^{lower}}) \text{: } I_{i,j}^{upper} = \begin{cases} e_j^{upper} \mathcal{S}_p & \text{if } e_i^{upper} \cdot \mathcal{S}_{p,1} < \overline{f_j^{upper}} - \overline{s_i^{lower}} \\ \overline{f_j^{upper}} - \overline{s_i^{lower}} & \text{otherwise} \end{cases} \tag{15}$$

$$\text{Case B } (\overline{s_j^{lower}} \geq \overline{s_i^{lower}}) \text{: } I_{i,j}^{upper} = \begin{cases} e_j^{upper} \mathcal{S}_p & \text{if } e_i^{upper} \cdot \mathcal{S}_{p,1} < \overline{f_i^{upper}} - \overline{s_j^{lower}} \\ 0 & \text{otherwise} \end{cases} \tag{16}$$

Note that in Eq.(15), we have chosen $\mathcal{S}_{p,1}$ to ensure that $e_i^{upper} \cdot \mathcal{S}_p$ is always less than $\overline{f_j^{upper}} - \overline{s_i^{lower}}$ for any choice of $\mathcal{S}_{p,q}$. Similarly, we have chosen $\mathcal{S}_{p,1}$ in the condition check $e_i^{upper} \cdot \mathcal{S}_{p,1} < \overline{f_i^{upper}} - \overline{s_j^{lower}}$ in Eq.(16).

By iteratively applying the above separation analysis and interference time analysis, we can derive upper bounds on process latencies with scaling factors being free variables. Since the latency bound $\mathcal{L}_k$ for each bottleneck process σ_k should be made no greater than the required latency $\mathcal{L}$, the latency constraint for $\mathcal{L}_k$ has the following form.

$$\mathcal{L}_k = g(\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_p, \ldots) \leq \mathcal{L} \tag{17}$$

Note that one or more equations are obtained for a single task if it has one or more immediate predecessors. This is because the **max** operation in Eq.(10) involves comparison of unknown values f_i^{upper}. Also note that all the equations Eq.(11) - Eq.(16) have linear forms with respect to the scaling factor $\mathcal{S}_p$. As a result, we end up with a set of linear constraints for each bottleneck process σ_k. The following example illustrates how to apply the analysis techniques and derive such linear constraints for the digital copier example.

Walk-through Example: Revert to the digital copier example. We derive latency constraints for the bottleneck processes σ_3 and σ_4 found in Section 3. The bottleneck processes σ_3 and σ_4 are hosted by three processing elements π_1, π_2, and π_3 as shown in Figure 5. Their scaling factors and corresponding cost values are given in Table 4.

As an illustration, we derive the latency constraints for τ_9. By applying Eq. (11) and (10), we can write

Table 4
Cost tables for π_1, π_2, and π_3.

$\mathcal{S}_{1,i_1}$	$\mathcal{S}_{1,1}$	$\mathcal{S}_{1,2}$	$\mathcal{S}_{1,3}$	$\mathcal{S}_{1,4}$
scaling factor	0.4	0.5	0.6	1.0
cost	100	50	20	0

Cost table of π_1

$\mathcal{S}_{2,i_2}$	$\mathcal{S}_{2,1}$	$\mathcal{S}_{2,2}$	$\mathcal{S}_{2,3}$	$\mathcal{S}_{2,4}$
scaling factor	0.5	0.6	0.8	1.0
cost	150	70	30	0

Cost table of π_2

$\mathcal{S}_{3,i_3}$	$\mathcal{S}_{3,1}$	$\mathcal{S}_{3,2}$	$\mathcal{S}_{3,3}$	$\mathcal{S}_{3,4}$	$\mathcal{S}_{3,5}$	$\mathcal{S}_{3,6}$
scaling factor	0.4	0.5	0.6	0.7	0.9	1.0
cost	300	200	150	100	50	0

Cost table of π_3

$$\begin{aligned} f_9^{upper} &= s_9^{upper} + I_9^{upper} + e_9^{upper} \cdot \mathcal{S}_1 \\ &= \max\{f_6^{upper}, f_7^{upper}, f_8^{upper}\} + I_9^{upper} + e_9^{upper} \cdot \mathcal{S}_1. \end{aligned} \tag{18}$$

To compute I_9^{upper} in the equation above, we first determine the worst-case execution window for each task using Eq.(12 - 13). We then apply the analysis techniques in Eq.(14 - 16) to find tight windows. After finding all the worst-case execution windows, we can obtain $\Phi(\tau_9) = \{\tau_3, \tau_{11}, \tau_{14}\}$ and $I_9^{upper} = (e_3^{upper} + e_{11}^{upper} + e_{14}^{upper}) \cdot \mathcal{S}_1$. Thus, the latency constraint for τ_9 is written as follows:

$$\begin{aligned} f_9^{upper} &= \max\{f_6^{upper}, f_7^{upper}, f_8^{upper}\} \\ &\quad + (e_3^{upper} + e_{11}^{upper} + e_{14}^{upper}) \cdot \mathcal{S}_1 + e_9^{upper} \cdot \mathcal{S}_1 \leq \mathcal{L}. \end{aligned}$$

Since the **max** operator in the equation above involves free variables, we can split the constraint to eliminate $\max\{\cdot\}$, as follows:

$$\begin{aligned} &f_6^{upper} + (e_3^{upper} + e_9^{upper} + e_{11}^{upper} + e_{14}^{upper}) \cdot \mathcal{S}_1 \leq \mathcal{L} \\ &f_7^{upper} + (e_3^{upper} + e_9^{upper} + e_{11}^{upper} + e_{14}^{upper}) \cdot \mathcal{S}_1 \leq \mathcal{L} \\ &f_8^{upper} + (e_3^{upper} + e_9^{upper} + e_{11}^{upper} + e_{14}^{upper}) \cdot \mathcal{S}_1 \leq \mathcal{L}. \end{aligned}$$

By recursively computing upper bounds on finish times of tasks in a reverse topological order, we have the following inequalities.

$$8\mathcal{S}_1 + 14\mathcal{S}_2 \leq 15,\ 12\mathcal{S}_1 + 8\mathcal{S}_2 \leq 15,\ 15\mathcal{S}_1 + 3\mathcal{S}_2 \leq 15,$$
$$17\mathcal{S}_1 \leq 15,\ 14\mathcal{S}_1 + 9\mathcal{S}_3 \leq 15,\ 8\mathcal{S}_1 + 11\mathcal{S}_3 \leq 15$$

4.2 Formulating Optimization Problem

After latency constraints are derived for bottleneck processes, we formulate an optimization problem with the objective of minimizing the total hardware cost given by $C(\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n) = \sum_{\pi_i \in PE} c_i(\mathcal{S}_i)$. Let Γ be an n-dimensional column vector of scaling factors $[\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n]^T$ that satisfies the latency constraints. Since every scaling factor $\mathcal{S}_i$ takes discrete values, each value can be represented by relabeling $\mathcal{S}_i$ with j_i in an increasing order. Let Λ be an n-dimensional column vector $[j_1, j_2, \ldots, j_n]^T$ that is associated with $[\mathcal{S}_{1,j_1}, \mathcal{S}_{2,j_2}, \ldots, \mathcal{S}_{n,j_n}]^T$ by $\Gamma = S(\Lambda)$. Suppose that we have n scaling factors and m latency constraints which are derived from our latency analysis. The problem can then be transformed into the following integer programming form:

minimize $C(S(\Lambda))$

subject to (latency constraint) $AS(\Lambda) \leq B$, and

subject to (range constraint) $L \leq \Lambda \leq U$,

where $C(S(\Lambda))$ decreases with respect to every j_i, A is an $m \times n$ matrix whose entries are the coefficients of scaling factors in the derived latency constraints, B is an $n \times 1$ matrix whose entries are the same as the required latency $\mathcal{L}$, L is an n-dimensional column vector whose entries are the lower bounds of j_i, and U is an n-dimensional column vector whose entries are the upper bounds of

j_i. Thus, $AS(\Lambda) \leq B$ is a matrix expression for the set of latency constraints and $L \leq \Lambda \leq U$ is a matrix expression for the range constraint of j_i.

Walk-through Example: Revert to the digital copier example in Section 2.2. The latency constraints derived in the previous section give the latency constraint matrices A and B:

$$A = \begin{pmatrix} 8 & 12 & 15 & 17 & 14 & 8 \\ 14 & 8 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 9 & 11 \end{pmatrix}^T \quad \text{and} \quad B = \begin{pmatrix} 15 & 15 & 15 & 15 & 15 & 15 \end{pmatrix}^T.$$

The cost function $C(\cdot)$ is determined by the cost tables given in Table 4. For example, $\Lambda = [2, 2, 3]^T$ maps to $C(S(\Lambda)) = C([0.5, 0.7, 0.6]^T) = 50+70+150 = 270$. The cost tables also give the range constraint matrices L and U;

$$L = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}^T \quad \text{and} \quad U = \begin{pmatrix} 4 & 4 & 6 \end{pmatrix}^T.$$

5 Heuristic Search Algorithm

The formulated optimization problem is an integer programming problem that is NP-hard. We may solve the problem by transforming it into a nonlinear programming problem and by applying nonlinear programming techniques such as the Lagrange method [15]. However, the solutions obtained by such techniques must be rounded off, thus yielding nearest integer suboptimal solutions. In many cases, rounding the results can even lead to infeasible solutions [15]. A possible approach to finding an optimal solution is to compute the cost of every feasible solution and select the one with the minimum cost. But this ap-

proach requires examining all possible combinations of scaling factors. If each scaling factor $\mathcal{S}_i$ can take m_i values, the search space is enormous possessing $m_1 \times m_2 \times \ldots \times m_n$ elements.

In this section, we present a heuristic search algorithm called the *k-level diagonal search* algorithm to solve the optimization problem. Since our algorithm employs the divide-and-conquer strategy, we first describe two heuristics used in that strategy. Then, we provide a complexity analysis for the algorithm.

5.1 Two Search Heuristics

The proposed algorithm employs two heuristics to find feasible solutions to the optimization problem while effectively reducing search time. The first heuristic exploits the monotonicity of the cost function: the cost increases as the scaling factor decreases. This leads to the geometric property that the local optimum is found at the end of a diagonal line traversing an n-dimensional rectangular parallelepiped (hyperbox) space. As shown in Figure 6 (A), the search starts from the origin and incrementally examines each point along the diagonal until any of the linear constraints is violated. This diagonal search can greatly reduce the search time since it explores a one-dimensional line without visiting all possible points in the n-dimensional hyperbox space.

The diagonal search can, however, examine only a hyperbox space while the problem space is arbitrary. To cover the entire space, the outer space of the hyperbox is split into disjoint subspaces and the diagonal search is iteratively applied to each subspace. The second heuristic makes use of the tangent planes of the hyperbox to partition the outer space of the hyperbox. As shown in

Figure 6 (B), the remaining space is cut by each tangent plane parallel to $\mathcal{S}_i = 0$, yielding n subspaces. Note that generated subproblem spaces are of the identical form as the original problem since each cutting plane is parallel to $\mathcal{S}_i = 0$.

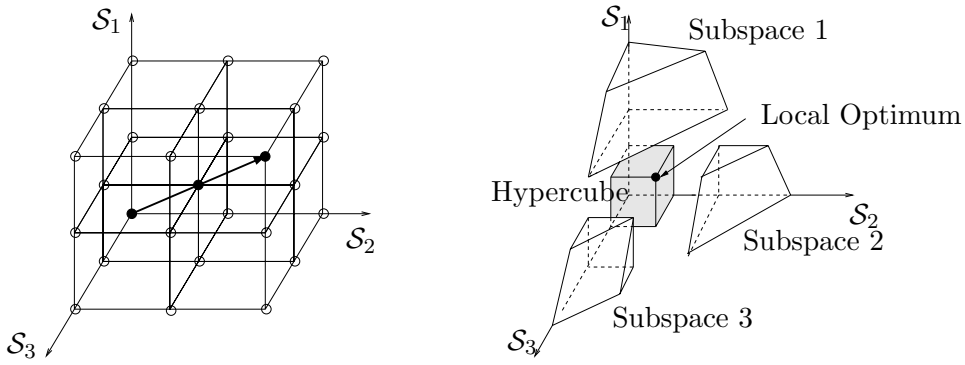


Fig. 6. (A) Diagonal search and (B) partition by tangent planes.

5.2 K-Level Diagonal Search Algorithm

Our algorithm iteratively applies the diagonal search and subproblem partitioning. This divide-and-conquer strategy generates a tree of subproblems, thus yielding several choices for the tree traversal policy. We use a breadth-first search that explores subproblems level by level in the problem tree since the breadth-first search allows us to trade between the search time and the optimality by limiting the search level: deeper searches will return more optimal solutions but will suffer from longer search times. We call this search algorithm k-level diagonal search because it limits the search depth to k levels.

Figure 7 shows the complete pseudocode of the k-level diagonal search algorithm. Let $G_{p,q}$ be the q^{th} subproblem space generated at the p^{th} level and let $L_{p,q} \leq \Lambda_{p,q} \leq U_{p,q}$ be the range constraint for $G_{p,q}$. For a given k and an

original problem space $G_{1,1}$, the algorithm applies diagonal search procedure P1 and subproblem partitioning procedure P2 to each subproblem space until it finds a k-level suboptimal solution. On line 6, the outermost loop increases the search level p from 1 to k in order to traverse the problem tree in the breadth-first order. On lines 9 to 26, the inner loop performs procedure P1 and P2 for each p^{th} level subproblem space $G_{p,q}$. Procedure P1 is described on lines 10 to 13. On lines 12 to 13, index vector Λ is increased in the direction of the unit diagonal vector $E = E_1 + E_2 + \ldots + E_n = [1, 1, \ldots, 1]$ and it is checked for the latency constraint $AS(\Lambda) \leq B$. This procedure finds the maximum increment $\delta_{p,q}E$ for the index vector Λ. Procedure P2 is described on lines 20 to 26. It calculates range constraint vectors L and U for each subspace. On lines 16 to 19, the algorithm calculates the minimum cost C^{min} among the local optimums that have been found so far.

There are two conditions for stopping subproblem generation at a problem node. The first condition is $\delta_{p,q} \not\geq 0$ (the condition on line 14). This implies that there exist no feasible solutions in $G_{p,q}$ and the algorithm does not generate any subproblems of $G_{p,q}$. The second condition is $L_{p,q} \not\leq U_{p,q}$, which indicates that $G_{p,q}$ is empty. The outermost loop terminates when p exceeds k or when no further subproblems are left (the condition on line 7). After terminating the outermost loop, the algorithm returns an optimal or a k-level suboptimal solution vector, if it has found one (lines 27 to 33).

We revert to our walk-through example. If we apply the k-level diagonal search to the integer programming problem found in Section 4, the algorithm stops at level 8 and the optimal solution is derived in subproblem $G_{3,6}$ at level 3.

Algorithm: K-Level Diagonal Search
begin
1: k: maximum search level
2: n: number of processing elements
3: G_p: set of p^{th} level subproblem spaces
4: **let** $G_1 = \{$the original problem space $G_{1,1}\}$
5: **let** $C^{min} = \infty$
6: **for** $p = 1$ to k
7: **if** G_p is empty
8: **break**
9: **for each** subproblem space $G_{p,q} \in G_p$
10: /* diagonal search */
11: $\delta_{p,q} = -1$
12: **while** $A\mathcal{S}(L_{p,q} + (\delta_{p,q} + 1)E) \leq B$ and $L_{p,q} + (\delta_{p,q} + 1)E \leq U_{p,q}$
13: $\delta_{p,q} = \delta_{p,q} + 1$
14: **if** $\delta_{p,q} < 0$
15: **continue**
16: $\Lambda = L_{p,q} + \delta_{p,q}E$
17: **if** $C(\mathcal{S}(\Lambda)) < C^{min}$
18: $\Gamma = \mathcal{S}(\Lambda)$
19: $C^{min} = C(\Gamma)$
20: /* subproblem partitioning */
21: $G_{p+1} = \phi$
22: **for** $j = 1$ to n
23: $L_{p+1,n(q-1)+j} = L_{p,q} + (\delta_{p,q} + 1)E_j$
24: $U_{p+1,n(q-1)+j} = U_{p,q} - \sum_{i=1}^{j-1}(U_{p,q}^T - L_{p,q}^T)E_iE_i + \delta_{p,q}\sum_{i=1}^{j-1} E_{p,q}^T E_iE_i$
25: **if** $L_{p+1,n(q-1)+j} \leq U_{p+1,n(q-1)+j}$
26: $G_{p+1} = G_{p+1} \cup \{G_{p+1,n(q-1)+j}\}$
27: **if** $C^{min} \neq \infty$
28: **if** G_p is empty
29: **return** Γ is an optimal solution vector
30: **else**
31: **return** Γ is a k-level suboptimal solution vector
32: **else**
33: **return** no feasible solution
end

Fig. 7. K-level diagonal search algorithm.

5.3 Analysis of K-Level Diagonal Search Algorithm

The running time of the k-level diagonal search algorithm largely depends on the number of checks for the latency constraints $A\mathcal{S}(\Lambda) \leq B$ (multiplication

and comparison of matrices). From this viewpoint, the approximate running time can be represented by $\sum_{p=1}^{k}\sum_{q=1}^{n^p}(\delta_{p,q}+1)$ where $(\delta_{p,q}+1)$ is the number of checks for latency constraints. The worst case occurs when every diagonal search fails to explore a hypercube space and examines only one point, i.e., $\delta_{p,q}=0$. This case generates the largest problem tree and the algorithm exhaustively examines candidates one by one. By substituting $\delta_{p,q}=0$, we can show that the complexity of the k-level diagonal search is $O(n^k)$. Although this is an exponential function of k, we can control the search time by limiting k. If search time is critical and a suboptimal solution is acceptable, we can choose a small value of k. While a large value for the search level k improves the solution, k is bounded from above if the problem space is finite. To compute the upper bound on k, we define a metric M that can capture the size of the problem space. Let $M_{p,q}$ be a metric for the size of the problem space $G_{p,q}$, which is defined as below.

$$M_{p,q} = ||U_{p,q} - L_{p,q}|| = (U_{p,q}^T - L_{p,q}^T)E \tag{19}$$

This metric $M_{p,q}$ determines the upper bound on k. One of the important advantages of the k-level diagonal search is that it always diminishes the metric value as the iteration proceeds, $M_{p,q} > M_{p+1,n(q-1)+r}$. The following theorem proves this assertion.

Theorem 1. *For metric $M_{p,q}$ of $G_{p,q}$ and metric $M_{p+1,n(q-1)+r}$ of any subproblem $G_{p+1,n(q-1)+r}$ generated by the k-level diagonal search algorithm,*

$$M_{p+1,n(q-1)+r} \leq M_{p,q} - (\delta_{p,q}+1) \tag{20}$$

Proof. From the definition of metric $M_{p,q}$ in Eq. (19), we have

$$M_{p+1,n(q-1)+r} = ||U_{p+1,n(q-1)+r} - L_{p+1,n(q-1)+r}||$$

By replacing $U_{p+1,n(q-1)+r}$ and $L_{p+1,n(q-1)+r}$ according to the k-level diagonal search procedure P2, and using properties $||A+B|| = ||A||+||B||$ and $||E_i|| = 1$, we obtain

$$\begin{aligned} M_{p+1,n(q-1)+r} &= ||U_{p,q} - \sum_{i=1}^{r-1}(U_{p,q}^T - L_{p,q}^T)E_iE_i + \delta_{p,q}\sum_{i=1}^{r-1}E_i - L_{p,q} - (\delta_{p,q}+1)E_r|| \\ &= ||U_{p,q} - L_{p,q}|| - \sum_{i=1}^{r-1}(||(U_{p,q}^T - L_{p,q}^T)E_iE_i|| - \delta_{p,q}) - (\delta_{p,q}+1) \end{aligned}$$

Since $||(U_{p,q}^T - L_{p,q}^T)E_iE_i|| - \delta_{p,q} \geq 0$, we get the inequality (20). □

According to the Theorem 1, the metric always decreases by at least one as the search level increases. The k-level diagonal search eventually terminates when metric $M_{p,q}$ becomes zero, which implies an empty space where $L_{p,q} = U_{p,q}$. Thus, the upper bound on k for the original problem $G_{1,1}$ is $(M_{1,1}+1)$, since a problem with a zero metric requires at least one level of search. We refer to this upper bound as a *guarantee level* in that it guarantees finding the optimal solution if the search level is increased up to $(M_{1,1}+1)$. However, the k-level diagonal search algorithm can terminate earlier than the guarantee level if $\delta_{p,q}$ is not zero. If $\delta_{p,q}$ is 1 at every search level, the algorithm terminates at a depth of half the guarantee level.

6 Performance Evaluation

We have evaluated the effectiveness of the proposed search algorithm through a series of simulations with randomly generated workloads. For performance comparison, we have implemented the k-level diagonal search algorithm (KD) and a brute-force search algorithm (BF). The BF algorithm exhaustively examines all candidate solutions.

Test problem sets were synthesized based on a random number generator for distinct dimension variables (the number of PEs) ranging from 3 to 8. It is necessary to limit the number of PEs below 9 since the BF algorithm needs to perform an extremely time consuming exhaustive search. For each number of PEs, the problem sets were generated as below.

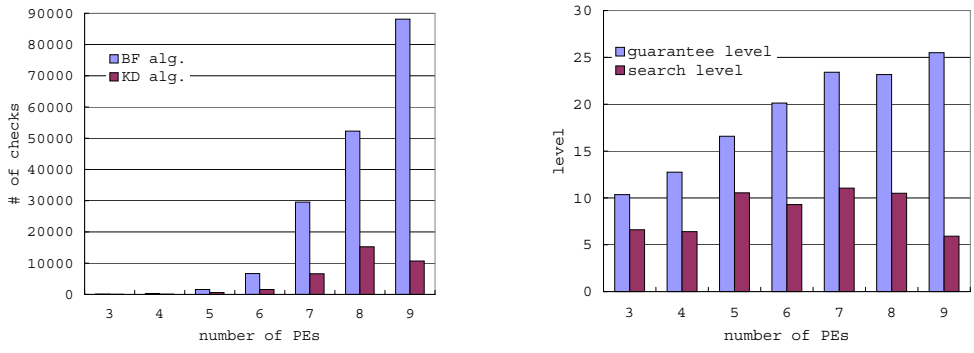
- The entries of latency constraint matrices (A and B) were chosen in range $[1, 1000000]$.
- Cost tables were obtained by sorting random numbers in range $[1, 100]$ to meet the monotonicity condition.

To assess the average case performance, two performance metrics were used: (1) the number of checks for latency constraints and (2) the maximum search level for finding optimal solutions. We measured the two performance metrics while running the three algorithms.

- The first performance metric was chosen as a measure to represent the running time of the algorithm. Figure 8 (A) shows that the KD algorithm greatly outperforms the BF algorithm. While the running time of the BF algorithm exponentially increases as the workload gets heavier, the running

time of the KD algorithm increase much more slowly and even decreases when the number of PEs exceeds eight. This anomaly can occur because the performance of the KD algorithm depends more on the shape of the feasible solution space than on the size of the problem. Due to the diagonal search strategy, the KD algorithm finds the optimal solution faster if the shape of the feasible space is closer to a hyperbox.

- The second performance metric is the maximum search level required for a complete search. Figure 8 (B) shows the comparison between the maximum search levels of the KD algorithm and the guarantee levels. Guarantee levels were computed by using Eq. (19). On the average, the actual search levels of the KD algorithm were around half of the guarantee level. This confirms our analysis result that the search level cannot exceed the guarantee level. The search level does not necessarily increase as the number of PEs increases. When the number of PEs is nine, the KD algorithm finds an optimal solution with a shallower search than when the number of PEs is eight. This result also support our assertion that the performance of the KD algorithm depends dominantly on the shape of the feasible space.



(A) (B)

Fig. 8. (A) Performance comparison between KD algorithm and BF algorithm and (B) guarantee level by KD algorithm vs. maximum search level.

7 Conclusions

We have formulated the problem of distributed embedded system re-engineering as performance optimization of the system and presented a systematic solution approach. The proposed formulation adopts general distributed embedded system model where fixed priority preemptive scheduling is used. Since preemptive scheduling incurs arbitrary interleaving of task instances, it is extremely difficult to estimate latencies of processes. Motivated by the work done by Yen and Wolf [27], we have proposed an algorithm that can yield tight latency bounds by iteratively applying separation analysis and interference time analysis. Our algorithm outperforms [27].

The proposed solution approach pinpoints performance bottlenecks of the system and selectively upgrades processing elements at the least cost. Unlike other re-engineering approaches based on critical path optimization, our approach does not lead to excessive optimization since it relies on the trade-off analysis between cost and performance. It appears that our approach can be employed even at the early stage of system design due to its capability of prototyping and systematic global latency analysis.

Although in this paper we have primarily focused on hardware upgrades, our formulation can be easily generalized to incorporate software upgrades when appropriate cost tables are given. In particular, our approach is well suited to the component-based design of an application-specific embedded system in that a system is built by composing software components selected from a predefined component library. We are currently extending our approach to designing a component-based real-time operating system and the result is

promising.

References

[1] I. Ahmad and Y.-K. Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 9(9):872–892, September 1998.

[2] P. Ancilotti, G. Buttazzo, M. Di Natale, and M. Spuri. A development environment for hard real-time applications. *International Journal of Software Engineering and Knowledge Engineering*, 6(3):331–354, 1996.

[3] R. Arnold, F. Mueller, and D. Whalley. Bounding worst-case instruction cache performance. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 172–181, December 1994.

[4] R. Bettati. *End-to-End Scheduling to Meet Deadlines in Distributed Systems*. PhD thesis, University of Illinois, Urbana-Champaign, 1994.

[5] A. Burns and A. Wellings. HRT-HOOD: A structured design method for hard real-time systems. *Real-Time Systems*, 6(6):73–114, 1994.

[6] S. Chatterjee, K. Bradley, J. Madriz, J. Colquist, and J. Strosnider. SEW: A toolset for design and analysis of distributed real-time systems. In *Proceedings of RTAW 97*, 1997.

[7] K. Ghose, S. Aggarwal, P. Vasek, S. Chandra, A. Raghav, A. Ghosh, and D. Vogel. A toolkit for real-time software design, development, and evaluation. In *Proceedings of Euromicro Workshop on Real-Time Systems*, 1997.

[8] H. Gomaa. *Software Design Methods for Concurrent and Real-Time Systems*. Addison Wesley, 1996.

[9] I. Jacobson. Re-engineering of old systems to an object-oriented architecture. In *Proceedings of the ACM OOPSLA'92*, pages 340–350. ACM Press, October 1998.

[10] G. Kahn. The semantics of simple language for parallel processing. In *the IFIP Congress 74*, 1974.

[11] K. Karadimitriou, J. Tyler, and N. E. Brener. Reverse engineering and reengineering of a large serial system into a distributed-parallel version. In *Proceedings of the 1995 ACM Symposium on Applied Computing*, pages 191–197. ACM Press, February 1995.

[12] E. Lee and T. Parks. Dataflow process networks. *IEEE Proceedings*, 83(5):773–801, May 1995.

[13] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 166–171. IEEE Computer Society Press, December 1989.

[14] S. Lim, Y. Bae, C. Jang, B. Rhee, S. Min, C. Park, H. Shin, K. Park, S. Moon, and C. Kim. An accurate worst case timing analysis for risc processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[15] D. G. Luenberger. *Linear and Nonlinear Programming.* Addison Wesley, 1989.

[16] V. K. Madisetti, Y. K. Jung, M. H. Khan, J. Kim, and T. Finnessy. Reengineering legacy embedded systems. *IEEE Design and Test of Computers*, 16(2):38–47, April-June 1999.

[17] J. Park, M. Ryu, S. Hong, and L. Lo Bello. Rapid re-engineering of embedded real-time systems via cost-benefit analysis with k-level diagonal search. In

Proceedings of IEEE Real-Time Systems Symposium, pages 257–266, December 2001.

[18] M. Ryu and S. Hong. End-to-end design of distributed real-time systems. *Control Engineering Practice*, 6(1):93–102, January 1998.

[19] W. L. Scherlis. Small-scale structural reengineering of software. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 116–120. ACM Press, October 1996.

[20] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object Oriented Modeling*. Wiley, 1994.

[21] P. Stevens and R. Pooley. Systems reengineering patterns. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 17–23. ACM Press, November 1998.

[22] D. B. Stewart and G. Arora. A tool for analyzing and fine tuning the real-time properties of an embedded system. *IEEE Transactions on Software Engineering*, 29(4):311–326, April 2003.

[23] J. Sun. *Fixed-Priority End-to-End Scheduling in Distributed Real-Time Systems*. PhD thesis, University of Illinois, Urbana-Champaign, 1997.

[24] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 50(2):117–134, 1994.

[25] H. Tokuda and M. Kotera. A real-time toolset for the ARTS kernel. Technical Report CMU-CS-88-180, Carnegie Mellon University, October 1988.

[26] R. R. Tummala and V. K. Madisetti. System on chip or system on package. *IEEE Design and Test of Computers*, 16(2):48–56, April-June 1999.

[27] T.-Y. Yen and W. Wolf. Performance estimation for real-time distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(11):1125–1136, November 1998.

[28] Rajesh K. Gupta. Co-Synthesis of Hardware and Software for Digital Embedded Systems. *Kluwer Academic Publishers*, 1995.

[29] R. Ernst, J. Henkel, and T. Benner. Hardware-Software Co-Synthesis for Microcontrollers. *IEEE Design and Test of Computers*, 10(4):64–75, December 1993.